Spring 2023 - CS 5490 - Project Report

Gates Lamb | Sam Smith | Yifei Sun
{u1033920 | u0629883 | u1298569}@utah.edu

Introduction

When the theory of electronic file sharing was launched, it was just a government project partnered with a few universities. It was 1969 when the first file was shared over the existing telephone system from one computer to another. At that time, there were only four nodes in the system that spoke to each other using what the project called the ARPANET (a network of those four computers). Those four nodes were at the University of Utah, UCLA, UCSB, and Stanford. To make a long story short, the ARPANET is what became the internet, and today the internet has over 10 Billion devices connected to it.

As time went on, and more devices were introduced, this opened internet servers up to what is called a Layer 7 HTTP Denial of Service (L7DoS) attack, among many others. What an L7DoS attack does is sends a mass amount of HTTP requests to a target server that overloads the computer on the server until it is shut down and rendered unresponsive. Many of these attacks target an endpoint on the target server that may require more resources than that of a static response. For example, requesting the home page of a site tends to render more of a static file that does not require many resources on the backend. But, when requesting content that is more dynamic - like a search for flights and prices for a certain day, or requesting your personal medical records - it requires more resources on the backend to search for that content, and return it to the user. When an attacker targets this dynamic content in mass, it can very quickly saturate the bandwidth of the server and cause an outage.

With the internet now connecting billions of devices, and the ever-evolving state of malware, L7DoS attacks are now capable of coming from more than one device. This is called a Layer 7 Distributed DoS attack (L7DDoS). In the past, you could simply prevent a L7DoS attack by blocking an IP if you see a huge amount of traffic coming from it. But as attackers evolved, they began spreading malware that could use distant machines to send the same request to the same server. What this did is show to the server that all of the traffic was coming from thousands, and sometimes millions, of devices, and rendered IP rate-limiting useless.

Our project sets out to prevent both L7DoS and L7DDoS attacks in one of many ways. Primarily, we work to find patterns within the incoming traffic over a set time frame, and if that fingerprinted traffic exceeds the thresholds we have set, we begin blocking the requests that follow the malicious pattern. What this does is prevent the server from having to embark on the process of finding the content on the backend which the request is searching for, and stops the traffic in its tracks, regardless of its IP address. This saves the target server from a bandwidth overload and allows the server to continue responding to non-malicious requests.

Related Work

It is difficult to know details of related work as the specifics of how these techniques are implemented by security companies are often not public knowledge, but we do know most effective existing HTTP-based Denial of Service prevention methods make use of traffic profiling techniques such as identifying IP reputation, identifying abnormal activity, implementing security challenges(such as CAPTCHA) and the use of Web Application Firewalls(WAFs). Existing solutions often don’t make use of rate-limiting techniques as high-volume attacks are usually picked up by defensive techniques at lower levels of the OSI Model. Since the primary goal of preventing HTTP-based DoS attacks is to prevent requests from causing the use of excess compute power, such as database accesses, the prevention methods also closely reflect NIDS/NIPS. These techniques include comparing traffic to previous threat signatures, client classification/behavior analysis, and specification-based analysis.

Our techniques for prevention in our implementation don't necessarily differ from the above but attempt to include rate-limiting techniques in a specification-based analysis to block malicious incoming HTTP requests while minimizing false positives.

Adversary Model

In designing the adversary model for our stateful firewall system, we make several assumptions about the adversary's capabilities and goals. The adversary in this context is an attacker who aims to disrupt the target server by launching Layer 7 HTTP/HTTPS DoS or DDoS attacks. This adversary can originate from any geographical location and operate from a single system or command a network of compromised systems (botnets) for a distributed attack.

The adversary's primary attack method involves sending a large volume of HTTP requests to the target server. The types of requests can vary, including GET, POST, or a mix of both (other methods like DELETE, PATCH, HEAD are not in the scope of our system prototype, but the implementations should be similar to what we have already accomplished), and target different endpoints. The adversary targets endpoints that require substantial computational resources on the server side, which helps to ensure maximum disruption to the server's operation.

During a L7DDoS attack, the adversary uses multiple compromised systems to send requests, making IP-based rate limiting ineffective. This distributed nature of the attack also makes IP-based blocking less effective, as the traffic originates from a variety of sources. For example, an adversary may command a botnet of compromised machines to launch the attack, making it appear as if the traffic is coming from multiple legitimate users. Additionally, the adversary can focus on endpoints that involve heavy computations, such as querying the availability of hotel guest rooms, which will force the server to perform large volumes of database queries to provide up-to-date information. The adversary can also send large data with POST requests to specific endpoints, consuming additional server resources.

Further, the adversary can be persistent and capable of launching attacks over extended periods. They have the capability to adapt their attack patterns based on the observed defense mechanism. For instance, if the defense system blocks traffic based on IP reputation or a specific pattern, the adversary may alter the attack pattern or use IP address spoofing to circumvent the defense.

Our stateful firewall is designed to counter these capabilities of the adversary. It requires users to send certain HTTP headers to our server for validation (enforced at frontend), instead using IP addresses to make blocking decisions. This validation process can be embedded in the frontend code base and it is easily achievable with most popular web frameworks, making it convenient and secure for legitimate users while providing a robust defense against adversaries.

Methodology

As we thought out how to tackle this problem, we brought ourselves into the mindset of a hotel company like Marriott. We decided it would make sense to protect against only the requests that require more dynamic content, and allow a CDN to deliver all of the static content. So, our static home page will be delivered by a CDN and the request would never actually make it to our origin server. The dynamic endpoint we decided to protect would be our hotel search endpoint. Not only would a request like this render mostly dynamic content, but it would require a lot of backend computation to fetch the info being requested.

From there, our solution takes a stateful approach on the search endpoint. For every two seconds of traffic, we store a combination of three headers of the request, and the count of how many times we have seen that header combination, into a dictionary. For the headers, we decided to use ‘user-agent’, sec-ch-ua’, and ‘accept’. One key detail is that we assume our solution delivers a JavaScript bundle to the search page that would ensure the three headers would be present in every search request. Simply, if we do not see any of these headers, we block. All three headers are expected if the user is coming from a browser, and if our JS is working as expected.

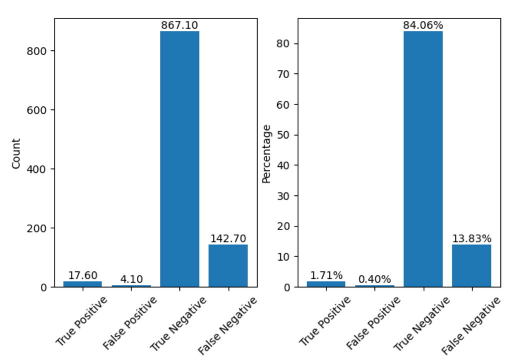
To be able to find a pattern and fingerprint an attack, our program counts the number of bytes included in the three headers. So, if the first request we see and the ‘user-agent’ had 21 bytes, sec-ch-ua’ had 34 bytes, and ‘accept’ had 16 bytes, we would store 71 in the dictionary with a count of 1. If we see another request with 71 bytes in the three headers, we increment the count to 2 for that header size, and so on. Every two seconds, our program looks at the dictionary of request size and count, and for any request sizes that have a count of 50 or more over the last two seconds, will be put into a blacklist. Then, before the server does anything with the search endpoint requests, it checks that blacklist to determine if the header size in the current request should be blocked or allowed through.

This is a simple, yet effective solution in that it protects the server compute, does not take long to start protecting against large amounts of similar traffic, and does not rely on the IP address of the request to block. Further, relying on three headers, instead of one or two, allows for more uniqueness in each request, which in turn will reduce false positives.

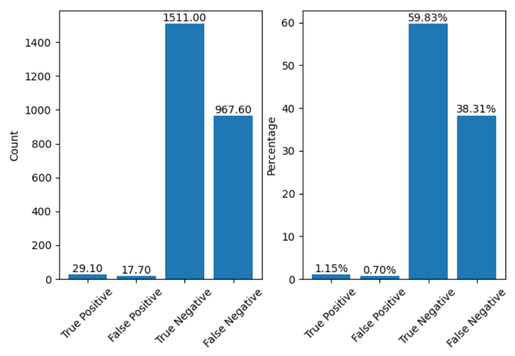
Implementations/Experimentations

Our experimentation primarily involved running our attack script against our server with differing values for the time window and the number of requests we receive before we begin to block. Our goal was to try and minimize false positives, which we found were inversely proportional to the size of the time window and the number of requests we see in that window before blocking. So after experimenting with varying values we settled on 2 seconds for the time window and 50 requests for the number of packets we see before blocking. Our final results show that even though we are performing little blocking overall our accuracy is quite good for smaller scale attacks. With an average of 85.8% accuracy for ~1000 requests and 61% for ~2500 requests. We achieve ~11% detection rate for ~1000 requests and ~3%. For false positives we achieve a rate of less than 0.5% at ~1000 requests and 1.2% at ~2500 requests.

Average of 1031.50 requests, completed in 2.30 seconds, 448.76 requests per second



Average of 2525.40 requests, completed in 2.39 seconds, 1056.07 requests per second



Currently our implementation suffers from prolonged attacks with greater number of requests, this is likely due to not having a proper way of refreshing our blacklist, so as time passes we will block more and more requests, both legitimate and malicious, but increasingly more legitimate traffic as the number of requests increases. The faster increase in blocking of legitimate traffic is due to legitimate traffic containing similar request sizes which over time more and more of that range would be blocked. Some solutions to this issue would be to only block during periods of high frequency in requests or to remove request sizes from our blacklist over time, this would allow more false negatives through but would make the implementation more effective at dealing larger scale longer attacks. With some tweaking to the implementation this style of prevention tool could be quite effective at passively blocking suspicious HTTP Post requests.

Conclusion

Given the landscape of the security world, the attacker will always find a way around a security solution. There will never be a 100% full-proof security solution, and ours surely isn't either. Our solution simply provides a different way of looking at Layer 7 data to prevent a DDoS attack while not relying on the IP address. We have shown it to be decently accurate and will certainly take out low-hanging fruit, all while accomplishing a low false positive rate.

Again, our solution is not perfect, and has weaknesses that are not easily patched. Although, if an attacker is successful in using a large amount of devices on the internet, and they launch similar traffic from each device, we can be successful in preventing that attack. If an attacker - for some reason - randomizes the three headers we look for, for every request, we would not be successful in blocking that. This back and forth can go on and on, but overall, we are happy with the results our solution yields, and we believe it is different from other solutions we have seen.

References

Source Code:
https://github.com/StepBroBD/CS5490-Project

What is an HTTP Flood | DDoS Attack Glossary | Imperva. (n.d.). Learning Center.
https://www.imperva.com/learn/ddos/http-flood/

NIDS/NIPS - Spring 2023 | Phillip Smith | In-Class Lecture for CS 5490/6490